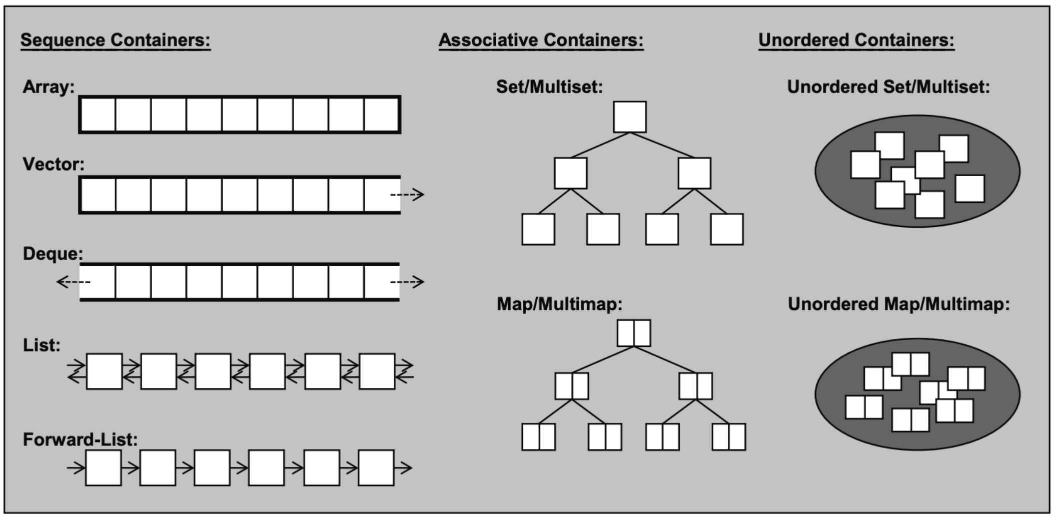
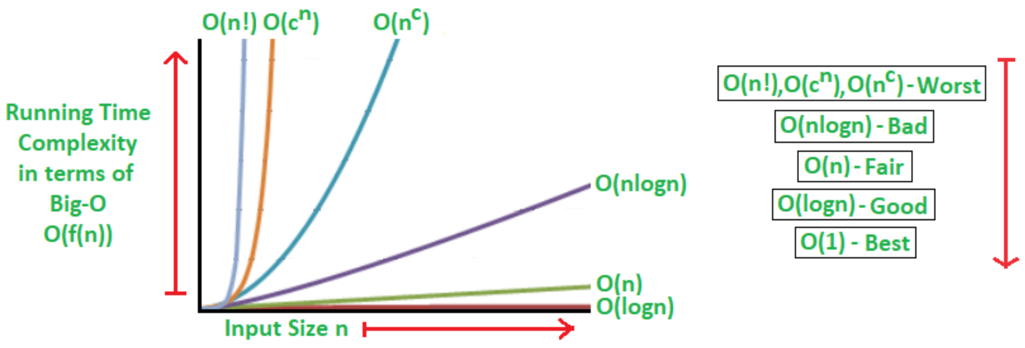
Containers

A container is an object that stores (in memory) a collection of other objects (elements). The containers and algorithms in the C++ Standard Template Library (STL) are implemented as class templates, and thus very flexible. The following are available STL containers. There are some nice tutorials on introducing the C++ STL[1].



Big O notation[2]

The Big O notation is used to describe how well an algorithm performs in terms of execution steps or memory size requirements terms of problem input size *n*. In other words, the cost for time or/and space. We also call this **complexity** of the algorithm. We also refer the complexity of a problem to be the best known algorithm for the problem: the complexity of the problem. For example, sorting of numbers with O(*n* log *n*).



[1] https://dev.to/pratikparvati/c-stl-containers-choose-your-containers-wisely-4lc4

[2] https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/

For a problem of size n:

- A constant complexity is O(1)
- A logarithmic complexity is O(log n)
- A linear complexity is O(n)
- A polynomial complexity is O(n^c)
- An exponential complexity is O(c^n)

Basic types of containers

- Sequences
    - Sequence containers implement data structures that can be accessed sequentially.
    - ordered collections in which every element has a certain position.
    - User controls the order of elements.
    - vector, list, deque
- Associative containers
    - Associative containers implement sorted data structures. Data can be quickly searched (O(log n) complexity).
    - sorted collections in which the position of an element depends on its key due to a certain sorting criterion
    - The container controls the position of elements within it.
    - When you insert, the container help you to sort the data
    - Elements can be accessed using a key.
    - set, multiset, map, multimap
- Unordered containers
    - Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).
    - the position of an element does not matter
    - unordered_set, unordered_multiset, unordered_map, unordered_multimap

Algorithms

STL also provides available algorithms to work with the containers[3]. Usually, the algorithms

[3] https://courses.cs.washington.edu/courses/cse333/18su/lectures/14-c++-STL.pdf

operate on the containers through iterators.

A set of functions to be used on ranges of elements

- **Range**: any sequence that can be accessed through iterators or pointers, like arrays or some of the containers

General form:

```
algorithm(begin, end, ...);
```

- Algorithms operate directly on range elements rather than the containers they live in
- Make use of elements' copy constructor, =, ==, !=, <
- Some do not modify elements
  - e.g. find, count, for_each, min_element, binary_search
- Some do modify elements
  - e.g. sort, transform, copy, swap

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void PrintOut(const int& p) {
    cout << p << " ";
}

int main() {
    vector<int> vec = {4,6,2,9,1,5,6,3};

    cout << "before sort: ";
    for_each(vec.begin(), vec.end(), &PrintOut);
    cout << endl;

    sort(vec.begin(), vec.end()); // O(n log n) complexity
    cout << "done sort: ";
    for_each(vec.begin(), vec.end(), &PrintOut);

    return 0;
}
```

Deque[4] (Sequential Container)

The term deque (it rhymes with "check") is an abbreviation for "**d**ouble-**e**nded **que**ue." It is a dynamic array that is implemented so that it can grow in both directions. Thus, inserting elements at the end and at the beginning is fast.

- Inserting elements at the end and at the beginning is fast (O(1) in complexity).
- However, inserting elements in the middle takes time because elements must be moved (O(n) in complexity).
- The elements of a deque are **not** stored contiguously: typical implementations use a sequence of individually allocated fixed-size arrays.
- Allocating deque contents from the center of the underlying array, and resizing the underlying array when either end is reached leads to frequent resizing and waste more space, particularly when elements are only inserted at one end.

It is important to note that, deque allocator allocates block of memory in a single allocation and hence frequent push_back() or push_front() has less memory allocation overhead compared to vector. In vector the memory is allocated in smaller chunks, hence frequent insertion of elements leads to frequent allocation of memory, slowing down the container.

```
#include <deque>
#include <iostream>

using namespace std;

int main() {
  deque < float >coll;
  for (int i = 1; i <= 6; ++i)
      coll.push_front(i * 1.1);    // insert at the front

  coll.push_back(10);

  for (int i = 0; i < coll.size (); ++i)
      cout << coll[i] << " ";

  cout << endl;
}
```

The push_front() function, however, is not provided for vectors, because it would have a bad runtime for vectors (if you insert an element at the front of a vector, all elements are moved).

4 https://en.cppreference.com/w/cpp/container/deque

Usually, the STL containers provide only those special member functions that in general have "good" performance, where "good" normally means constant or logarithmic complexity. This prevents a programmer from calling a function that might cause bad performance.

List[5] (Sequential Container)

A list is implemented as a doubly linked list of elements. This means each element in a list has its own segment of memory and refers to its predecessor and its successor with pointers.

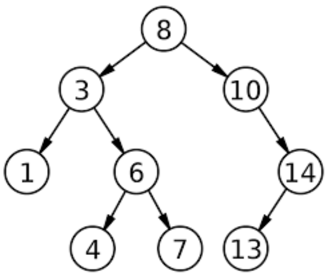
- Unlike vectors and deques, fast random access to list elements is not supported. To access the 10th element, you must navigate the first 9 elements by following the chain of their links resulting to O(n) linear complexity.
- list supports bidirectional iterators and allows constant time O(1) insert and erase operations anywhere within the sequence.
- storage management handled automatically
- If only uni-directional list traversal is needed, std::forward_list can be used.

```
#include <algorithm>
#include <iostream>
#include <list>

using namespace std;

int main(){
    list<int> l = {17, 55, 16, 3};

    //cout << l[3];  // cannot do this!

    // Insert an integer before 16 by searching
    auto it = find(l.begin(), l.end(), 16); // it points at 16
    if (it != l.end()){
        l.insert(it, 77); // constant time insert
        // list is now 17, 55, 77, 16, 3
    }
    it--; // move it to 77
    it--; // move it to 55
    l.erase(it); // constant time erase

    for (const auto &ele : l)
        cout << ele << ' ';
    cout << std::endl;
}
```

[5] https://en.cppreference.com/w/cpp/container/list

Set[6] (Associative Container)

A set is a collection in which elements are sorted according to their own values. Each element may occur only once, thus duplicates are not allowed.



- Insert and erase is O(log *n*) complexity.
- Initialize with unsorted sequence is O(*n* log *n*) complexity.
- Initialize with sorted sequence is O(*n*) complexity.
- lower_bound and upper_bound can be used with set.
    - NOTE: lower_bound(st.begin(), st.end(), x) from the STL algorithm will compile but is O(*n*) for std::set, while the member function lower_bound() of std::set has O(log *n*) complexity. Choose wisely[7].

```
#include <iostream>
#include <set>

using namespace std;

void printList(const set<int>& st){
    for (const auto &ele : st)
        cout << ele << ' ';
    cout << endl;
}

int main() {
    std::set<int> st = {9,3,12,2,10,4};
    printList(st);

    st.insert(1);
    st.insert(1); // another 1
    st.insert(6);
    st.insert(11);
    printList(st);
}
```

Multiset[8] (Associative Container)

A multiset is the same as a set except that duplicates are allowed. Thus, a multiset may contain multiple elements that have the same value.

[6] https://en.cppreference.com/w/cpp/container/set

[7] Algorithm lower_bound https://en.cppreference.com/w/cpp/algorithm/lower_bound
Set::lower_bound https://en.cppreference.com/w/cpp/container/set/lower_bound

[8] https://en.cppreference.com/w/cpp/container/multiset

You can change the code for set to multiset for an example.

The time complexity of count() is additionally linear (O(n)) to the number of elements having the same key.

Map[9] (Associative Container)

A map contains elements that are key/value pairs. Each element has a key that is the basis for the sorting criterion and a value. Each key may occur only once, thus duplicate keys are not allowed. A map can also be used as an associative array, which is an array that has an arbitrary index type.

- Since multiset count() could be slow, we can use map to store the frequency of a key instead of inserting multiple copies of they same key into a multiset .
- Insertion, deletion and lookup has logarithmic complexity O(log *n*).
- If the index operator is called with a non-existing number, it stores this number in the map and uses the default constructor for generating the data. This ensures that the index operator never returns an invalid reference.
- In order to prevent above case, find() should be called beforehand.
-

We use pairs in maps and multimaps. To create a pair, one way is to use the make_pair() function[10]. You can also place your pair of data in braces such as {object1,object2}.

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

int main(){
    map<int, string> mp = { make_pair(7, "I love C++!"),
                            make_pair(8, "I love OOP!") };

    mp.insert({3, "associative"});
    mp.insert(make_pair(1, "map"));
    mp.insert(make_pair(4, "container"));
```

[9] https://en.cppreference.com/w/cpp/container/map

[10] https://en.cppreference.com/w/cpp/utility/pair/make_pair

```
    mp.insert(make_pair(2, "is an"));
    mp.insert(make_pair(1, "key")); // another pair with key 1

    for (const auto &ele : mp)
       cout << ele.first << " : " << ele.second << endl;

    if(mp.find(8) != mp.end())
       cout << mp[8] << endl;
    else cout << "no element with key 5" << endl;

    if(mp.find(5) != mp.end())
       cout << mp[5] << endl;
    else cout << "no element with key 5" << endl;
}
```

Multimap[11] (Associative Container)

multimap differs from map in the same way as multiset differs from set.

Customizing the Comparison

```
#include <iostream>
#include <set>

using namespace std;

struct Cmp {
   bool operator()(const int& lhs, const int& rhs) const {
      return lhs > rhs;
   }
};

int main () {
   auto comp =[](int a, int b)    // lambda expression
   { return a > b;};

   set<int, decltype (comp)> st1 = {34, 45, 1, 77, 98, 15};
   set<int, Cmp> st2 = {34, 45, 1, 77, 98, 15};
   set<int, greater<int>> st3 = {34, 45, 1, 77, 98, 15};
   cout << "With custom compare comp: ";
   for (const auto & ele:st2)
      cout << ele << ' ';
   cout << endl;
}
```

unordered_set[12], unordered_multiset, unordered_map, unordered_multimap (Unordered containers)

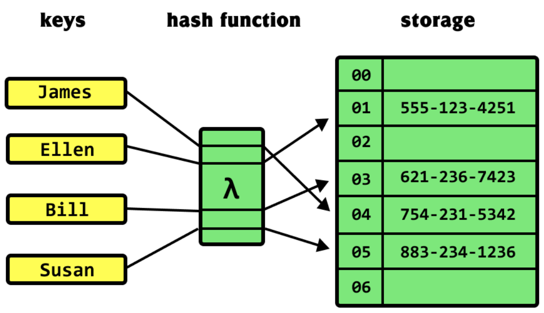
With hash tables, items can be found in O(1) time. The size of the hash table can be

[11] https://en.cppreference.com/w/cpp/container/multimap

[12] https://en.cppreference.com/w/cpp/container/unordered_set

manipulated by the user, and the hash function can also be chosen individually, which is important, because the performance versus space consumption characteristics depend on that.

- These containers has Hash table implementation
- Since elements are unordered, you can**not** perform binary search
- Also, begin() does **not** return the smallest element
- In order to use unordered_set/map, classes must have hash function implemented



```
#include <iostream>
#include <unordered_map>
using namespace std;

int main(){
    unordered_map<string, int> umap = {{"C++", 0}, {"NTU",100}};

    umap["OOP"] = 10;
    umap["Practice"] = 20;

    for (const auto& x : umap)
      cout << x.first << " " << x.second << endl;
}
```

You can also define your own hash function. Sometimes you have to do this, because the type you store in the unordered container has no default hash function. For example, storing pair of values as the key.

```
#include <iostream>
#include <unordered_set>

using namespace std;

struct Hasher {
    size_t operator()(const pair<int, int>& x) const {
        return x.first ^ x.second; // bitwise xor
    }
};

int main() {
    unordered_set<pair<int, int>, Hasher> us;
    us.insert({5, 2});
    us.insert({6, 1});
    us.insert({1, 7});
    us.insert({4, 9});
    us.insert({0, 6});

    for (const auto& x : us) {
      cout << x.first << " " << x.second;
      cout << " Hash value: " << us.hash_function()(x) << endl;
    }

    cout << us.bucket_size(6) << std::endl;
}
```

Another example when customizing hash function for two strings:

```
#include <iostream>
#include <unordered_map>

using namespace std;

struct Hasher {
    // We use predefined hash functions of strings and
    // define our hash function as XOR of the hash values.
    size_t operator()(const pair<string, string> &s) const {
        return (hash<string>()(s.first)) ^
                        (hash<string>()(s.second)); // bitwise xor
    }
};

int main() {
    unordered_map< pair<string, string>, string, Hasher> us;

    us.insert({{"C++", "OOP"}, "Great"});
    us.insert({{"NTU", "CE"}, "Wonderful"});

    for (const auto& x : us) {
      cout << x.first.first << " " << x.first.second << " is " <<
x.second;
      cout << ", Hash value: " << us.hash_function()(x.first) <<
endl;
    }
}
```

Adapter Containers

Container adapters are a special type of container class. They are not full container classes on their own, but wrappers around other container types (such as a vector, deque, or list). These container adapters encapsulate the underlying container type and limit the user interfaces accordingly.

- There are three standard container adapters: stacks, queues, and priority queues.
- Constructor adapters allow you to pass a specific allocator to their constructors.
- Container adapters provide the emplace() feature, which internally creates a new element initialized by the passed arguments

Stack[13] (Adapter Container)

A stack is a container which allows insertion, retrieving, and deletion only at one end (Last In First Out, LIFO). Objects inserted first are removed last. Any one of the sequence container could be used as a stack, if those containers supports the following operation.

- Insertion at one end (push, with push_back()).
- Deletion from the same end (pop, with pop_back()).
- Retrieving the value at that end (top, with back()).
- Testing the stack being empty (with empty()).
- default implementation using a deque.

```
#include <iostream>
#include <stack>

using namespace std;

int main(){
    std::stack<int> s;
    s.push(10);
    s.push(30);
    s.push(50);

    cout << "Size: " << s.size() << endl;
    cout << "Top: " << s.top() << endl;

    cout << "Elements of stack: ";

    while(!s.empty()){
        cout << s.top() << ' ';
        s.pop();
    }
}
```

[13] https://en.cppreference.com/w/cpp/container/stack

Queue[14] (Adapter Container)

A queue allows you to insert objects at one end and to remove them from the opposite end (First In First Out, FIFO). Objects at both ends can be read without being removed.

- list and deque are suitable data types for queue implementation.
- The queue has a default implementation using a list.
- push() inserts any number of elements.
- pop() removes the elements in the same order in which they are inserted.
- front() returns the first element in the queue.
- back() returns the last element in the queue.

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
    std::queue<int> q;
    q.push(10);
    q.push(30);
    q.push(50);

    cout << "Size: " << q.size() << endl;
    cout << "Front: " << q.front() << endl;
    cout << "Back: " << q.back() << endl;

    std::cout << "Elements of queue: ";

    while(!q.empty()){
        cout << q.front() << ' ';
        q.pop();
    }
}
```

Priority_queue[15] (Adapter Container)

It always returns the element with the highest priority using its top(). The pop() removes the top. The priority criterion must be specified. Usually it is the greatest (or smallest) number in the queue. We can use custom compare to describe the priority (default is std::less<int>).

The priority queue provides O(1) constant time lookup top() of the largest (by default)

[14] https://en.cppreference.com/w/cpp/container/queue

[15] https://en.cppreference.com/w/cpp/container/priority_queue

element, at the expense of O(log *n*) logarithmic time insertion push() and extraction pop().

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
    //priority_queue<int, vector<int>, greater<int>> pq;
    priority_queue<int> pq;
    pq.push(10);
    pq.push(30);
    pq.push(50);

    cout << "Size: " << pq.size() << std::endl;
    cout << "Top: " << pq.top() << std::endl;

    cout << "Elements of queue: ";

    while (!pq.empty()){
        cout << pq.top() << ' ';
        pq.pop();
    }
}
```

How to choose the right container for your purpose?[16]

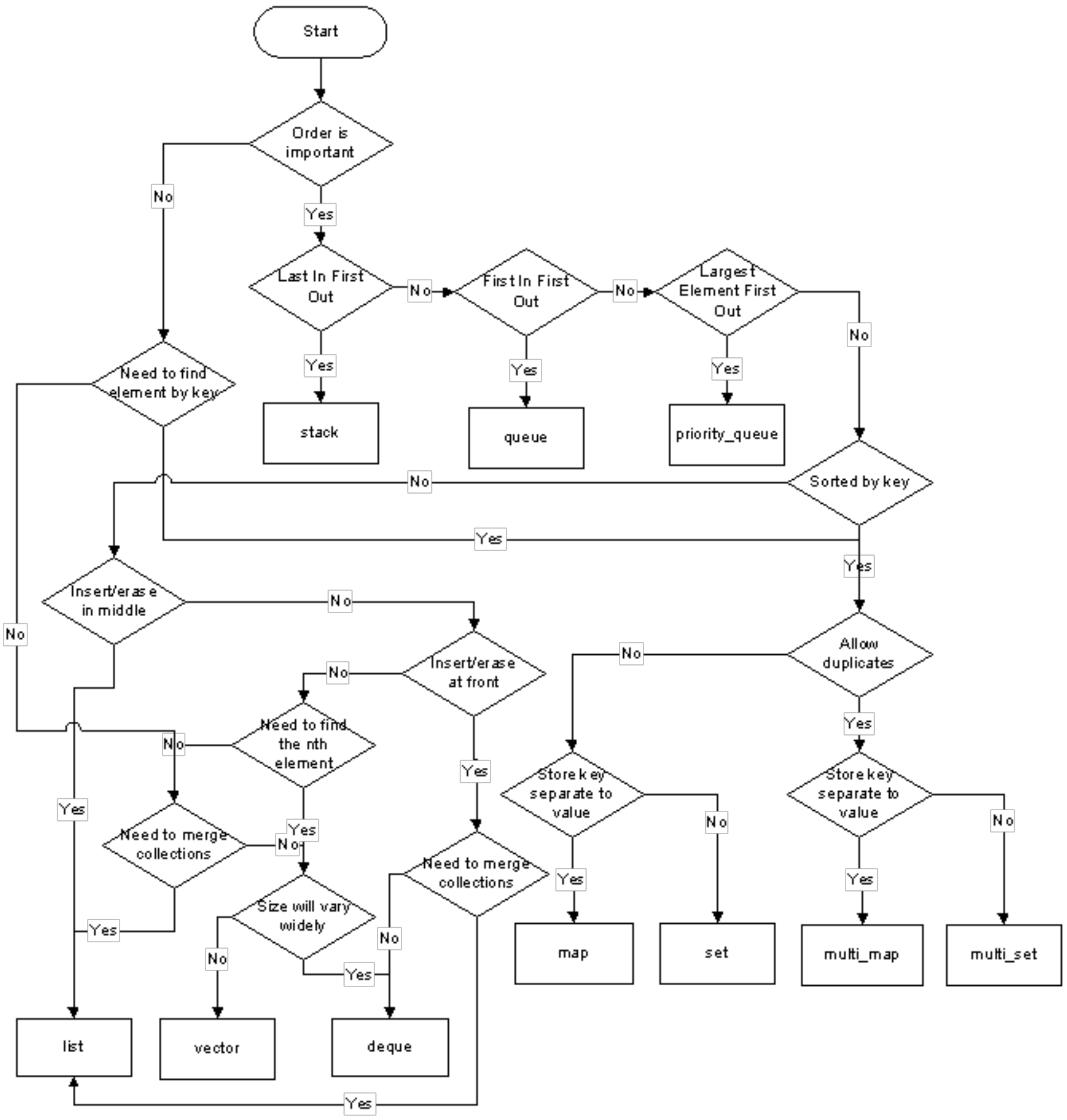
- Do you need to be able to insert a new element at an arbitrary position in the container? - if so, you need a sequence containers.
    - Size known: array
    - Size may chance: vector
    - If you insert and/or remove elements often at the beginning and the end of a sequence, you should use a deque. You should also use a deque if it is important that the amount of internal memory used by the container shrinks when elements are removed. Also, because a vector usually uses one block of memory for its elements, a deque might be able to contain more elements because it uses several blocks.
    - Need to avoid movement of existing containers elements when insertions and erasures take place? – yes, avoid contiguous containers (array, vector, deque)
    - If there are frequent insertion and remove of the elements in the middle of the sequence- use std::list instead of vector and deque. It does those operations in

[16] https://www.hackerearth.com/practice/notes/c-stls-when-to-use-which-stl/

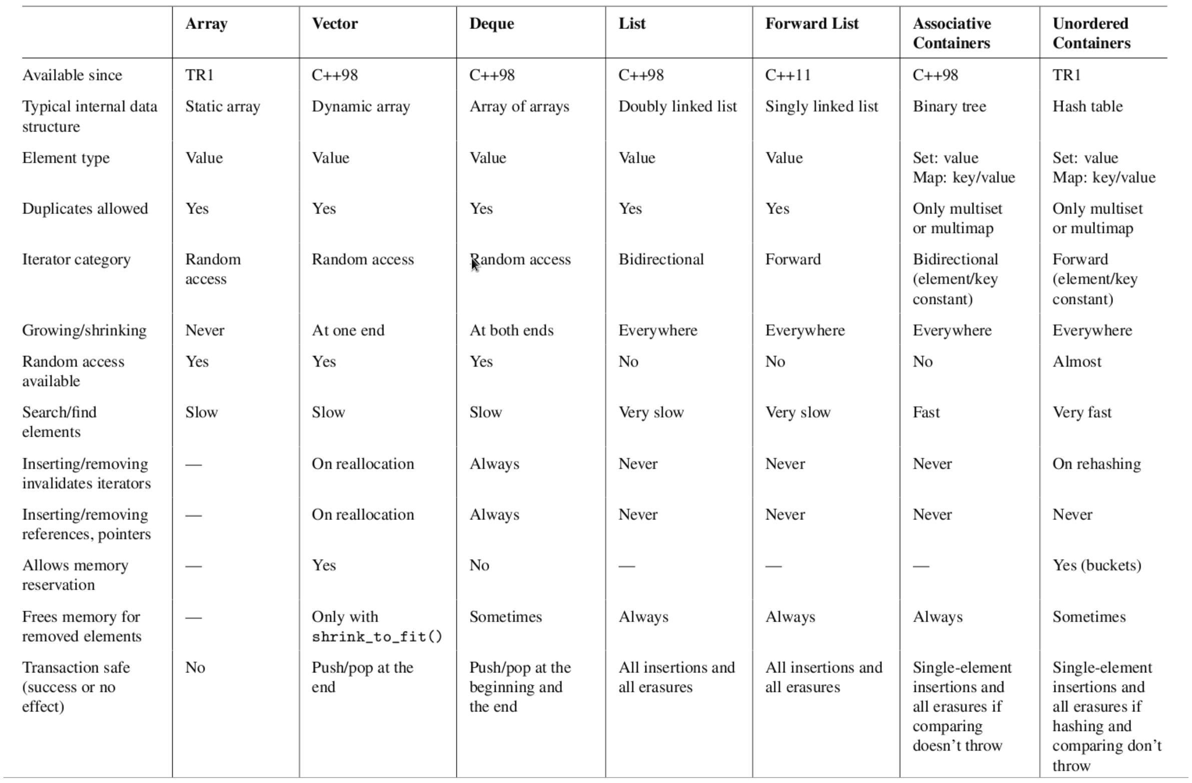
O(1) constant time by using list.

- If you often need to search for elements according to a certain criterion, use an unordered_set or unordered_multiset that hashes according to this criterion. However, hash containers have no ordering, so if you need to rely on element order, you should use a set or a multiset that sorts elements according to the search criterion.
    - If there are strict requirements on memory usage, additional memory overhead for storing hash table (unordered_set, unordered_map) cannot be accepted.
    - To process key/value pairs, use an unordered (multi)map or, if the element order matters, a (multi) map.

The following chart can help determine which container to use.



The following table lists some details about each container:



	Array	**Vector**	**Deque**	**List**	**Forward List**	**Associative Containers**	**Unordered Containers**
Available since	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Typical internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree	Hash table
Element type	Value	Value	Value	Value	Value	Set: value Map: key/value	Set: value Map: key/value
Duplicates allowed	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)	Forward (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere	Everywhere
Random access available	Yes	Yes	Yes	No	No	No	Almost
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast	Very fast
Inserting/removing invalidates iterators	—	On reallocation	Always	Never	Never	Never	On rehashing
Inserting/removing references, pointers	—	On reallocation	Always	Never	Never	Never	Never
Allows memory reservation	—	Yes	No	—	—	—	Yes (buckets)
Frees memory for removed elements	—	Only with `shrink_to_fit()`	Sometimes	Always	Always	Always	Sometimes
Transaction safe (success or no effect)	No	Push/pop at the end	Push/pop at the beginning and the end	All insertions and all erasures	All insertions and all erasures	Single-element insertions and all erasures if comparing doesn't throw	Single-element insertions and all erasures if hashing and comparing don't throw

Iterators could be invalidated when the content in the container changes. Like all node-based containers, a list doesn't invalidate iterators that refer to elements, as long as those elements are part of the container. Vectors invalidate all their iterators, pointers, and references whenever they exceed their capacity and part of their iterators, pointers, and references on insertions and deletions. Deques invalidate iterators, pointers, and references when they change their size, respectively. We might go into more details later in this semester about this issue.

If you think the underlying logic of these containers are interesting, you can consider taking a course such as data structures and algorithms.